CSEE 4840
Embedded System Design Lab 1: Using the FPGA

Stephen A. Edwards, Columbia University

Spring 2024

Learn how to code in SystemVerilog, run the Verilator simulator, observe simulated waveforms with GTKWave, and compile and download an FPGA-only project to the DE1-SoC board. You will create a system that can test the Collatz conjecture over a small range of values.

For this lab, you will use the open-source Verilator SystemVerilog simulator, the open-source GTKWave waveform viewer, and Quartus Prime 21.1 FPGA design software produced by Intel for their chips. You can find this software on the workstations in 1235 Mudd (on which you should have an account if you registered for the class), you may be able to run it on your laptop, or use some combination of both. Quartus Prime 21.1 Lite is free to download from the Intel website https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime/resource.html, although you will have to register for a free account and it only runs under Windows or Linux. While there are newer versions available, we suggest you stick to 21.1 for consistency.

To submit this assignment do two things:

1. Put your SystemVerilog code files (hex7seg.sv, collatz.sv, range.sv, and lab1.sv) into a .tar.gz file (e.g., `make lab1.tar.gz`) and upload it to Courseworks.

2. Demonstrate your working system to a TA. See Section 10 for details.

1 Download and Unpack the Lab 1 files

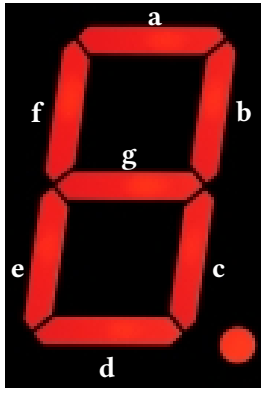
Download *lab1.tar.gz* from the class website and extract it by typing *tar zxf lab1.tar.gz*. This will create a *lab1* directory containing the files listed below.

Name	Contents
Makefile	Commands for creating the project files, compiling the project, building the *lab1.tar.gz* file, and cleaning up unneeded files.
hex7seg.sv	A module skeleton for a hex-to-seven segment decoder for displaying numbers on the board
collatz.sv	A module skeleton for computing the Collatz iteration for a particular number.
range.sv	A module skeleton for computing the Collatz iteration over a range of numbers and storing the iteration counts in a small memory.
lab1.sv	A module skeleton that provides a user interface to the modules above.
hex7seg.cpp	A Verilator test bench for the hex7seg module.
collatz.cpp	A Verilator test bench for the collatz module, which, when working, prints a Collatz sequence from a particular value.
range.cpp	A Verilator test bench for the range module, which, when working, runs the collatz module over a range of numbers and stores the result in a small memory.
collatz.gtkw	A GTKWave "save" file that remembers what signals to display, etc. for the collatz module.
range.gtkw	A GTKWave "save" file for the range module.
range-done.gtkw	A GTKWave "save" file that displays what should be the end behavior of the range module.
de1-soc-project.tcl	A Tcl script that creates the lab1 project files. Includes pin assignments.

Modify the hex7seg.sv, collatz.sv, range.sv, and lab1.sv files.

You may modify any other files; we will not grade them.

2 Implement and Test a Hex-to-Seven-Segment Decoder



The DE1-SoC includes six seven-segment displays, which we will use to display hexadecimal numbers. Each segment is connected to its own pin. These segment signals are active-low: a "0" turns them on. Bit 0 (the rightmost) of each 7-bit segment vector is the "a" segment, bit 1 is the "b" segment, etc., up to bit 6, the leftmost, which controls the "g" segment.

The *hex7seg.sv* file includes the interface to this module:

```
module hex7seg( input logic  [3:0] a,
                output logic [6:0] y );
```

Implement the body of the sevent-segment decoder module in *hex7seg.sv*. We have provided a Verilator testbench to test your implementation. Make sure Verilator is installed and compile and run the simulation:

```
$ make hex7seg
0 40 OK
1 79 OK
2 24 OK
3 30 OK
4 19 OK
5 12 OK
6 02 OK
7 78 OK
8 00 OK
9 10 OK
a 08 OK
b 03 OK
c 46 OK
d 21 OK
e 06 OK
f 0e OK
SUCCESS
```

which shows the 16 possible inputs and their outputs. An incorrect output will produce

```
7 7a INCORRECT expected 78
8 00 OK
FAILED
```

Run *make lint* to have Verilator run a fast, thorough check on your SystemVerilog code. The solution you eventually submit should not report any lint errors.

3 The Collatz Conjecture

The Collatz Conjecture is that for any positive integer n, $f^k(n) = 1$ for some positive integer k, where

$$f(n) = \begin{cases} n/2, & \text{if } n \text{ is even;} \\ 3n+1 & \text{otherwise,} \end{cases}$$

and $f^k(n)$ means to apply the function f k times: $f^k(n) = \underbrace{f(f(\cdots f(n)\cdots))}_{k \text{ times}}$.

For example, for $n = 5$, the sequence is

$$5\ 16\ 8\ 4\ 2\ 1,$$

and for $n = 7$, the sequence is

$$7\ 22\ 11\ 34\ 17\ 52\ 26\ 13\ 40\ 20\ 10\ 5\ 16\ 8\ 4\ 2\ 1.$$

The number of iterations it takes to reach 1 varies erratically. Here is a list of various n and the number of iterations required for that n. These number are in hexadecimal, which you will eventually display on the DE1-SoC.

```
 7 11    17 10    27 23      f7 30     3ff 3f    40f 3f     4ef b1
 8  4    18  b    28  9      f8 6e     400  b    410 20     4f0 28
 9 14    19 18    29 6e      f9 30     401 25    411 7d     4f1 28
 a  7    1a  b    2a  9      fa 6e     402 25    412 7d     4f2 20
 b  f    1b 70    2b 1e      fb 42     403 25    413 7d     4f3 20
 c  a    1c 13    2c 11      fc 6e     404 7d    414 20     4f4 28
 d  a    1d 13    2d 11      fd 6e     405 7d    415 20     4f5 28
 e 12    1e 13    2e 11      fe 30     406 7d    416 7d     4f6 20
 f 12    1f 6b    2f 69      ff 30     407 25    417 7d     4f7 20
10  5    20  6    30  c     100  9     408 7d    418 20     4f8 3a
11  d    21 1b    31 19     101 7b     409 9c    419 3f     4f9 20
12 15    22  e    32 19     102 7b     40a 7d    41a 20     4fa 3a
13 15    23  e    33 19     103 7b     40b 7d    41b 5e     4fb 54
14  8    24 16    34  c     104 1e     40c 7d    41c 51     4fc 3a
15  8    25 16    35  c     105 1e     40d 7d    41d 51     4fd 3a
16 10    26 16    36 71     106 1e     40e 3f    41e 51     4fe 85
```

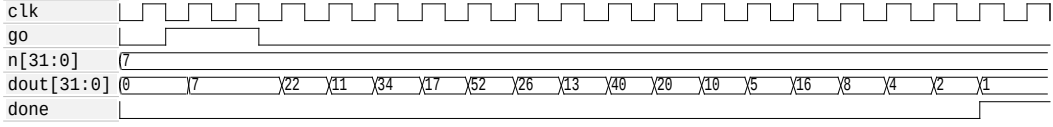
4 Write and Test a Collatz Sequence Generator

Implement a module that can test the Collatz conjecture for a particular *n* by completing the body of the provided *collatz* module in *collatz.sv*. Its interface is

```
module collatz( input logic        clk,  // Clock
                input logic        go,   // Load value from n; start iterating
                input logic [31:0] n,    // Start value; only read when go = 1
                output logic [31:0] dout, // Iteration value: true after go = 1
                output logic       done); // True when dout reaches 1
```

In every cycle, if *go* is true, the module should reset and start counting from the 32-bit unsigned integer value on *n*. The *dout* signal should always output the current value. In every other clock cycle, the module should compute the next number in the sequence by checking whether the current number is positive and either divide by two or multiply by three and add one. When the value reaches 1, *done* should be asserted and the module should stop producing new numbers until the next *go* input.

Here is the correct output running with the input 7. Note that *dout* is loaded with 7 starting at the first rising edge of the clock where *go* is asserted, but 22, the second number in the sequence, only appears in the first cycle after *go* is asserted.



We have provided a Verilator test bench that supplies appropriate inputs to the *collatz* module (i.e., *clk*, *go*, and *n*). To compile your *collatz.sv* file with the provided test bench into a Verilator simulator, run *make obj_dir/Vcollatz* or

```
verilator -trace -Wall -cc collatz.sv -exe collatz.cpp -top-module collatz
cd obj_dir
make -j -f Vcollatz.mk
```

Run the Verilator simulator for this module by typing *make collatz.vcd* or *./obj_dir/Vcollatz*, which prints the sequence it finds. When working, it prints

```
$ ./obj_dir/Vcollatz
7 22 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1
```

As a side-effect, running the *Vcollatz* simulation produces a Value Change Dump (VCD) file *collatz.vcd*, which you can view with the *gtkwave* program. Invoke it with *gtkwave collatz.vcd*. A "save" file controls which signals are displayed. You may use the one provided with *gtkwave --save=collatz.gtkw collatz.vcd*.

5 Write and Test a Module That Checks a Range of Values

Write a module that uses your Collatz sequence generator to test the Collatz hypothesis for a sequence of numbers and record the number of iterations each took. Fill in the body of the *range* module provided in *range.sv*. Its interface is

```
module range
    #(parameter
      RAM_WORDS = 16,            // Number of counts to store in RAM
      RAM_ADDR_BITS = 4)         // Number of RAM address bits
    (input logic              clk,    // Clock
     input logic              go,     // Read start and start testing
     input logic    [31:0]    start,  // Number to start from or count to read
     output logic             done,   // True once memory is filled
     output logic   [15:0]    count); // Iteration count data once finished
```

The two compile-time parameters *RAM_WORDS* and *RAM_ADDR_BITS* set the size of the memory in which the iteration counts should be stored.

The *go* signal should tell the module to read the *start* input and start generating Collatz iterations from that number. The number of iterations it takes to reach 1 starting from *start* should be written into address 0 in RAM; the number of iterations from *start* + 1 should be written into address 1, etc. Finally, *done* should be asserted when the RAM is filled.

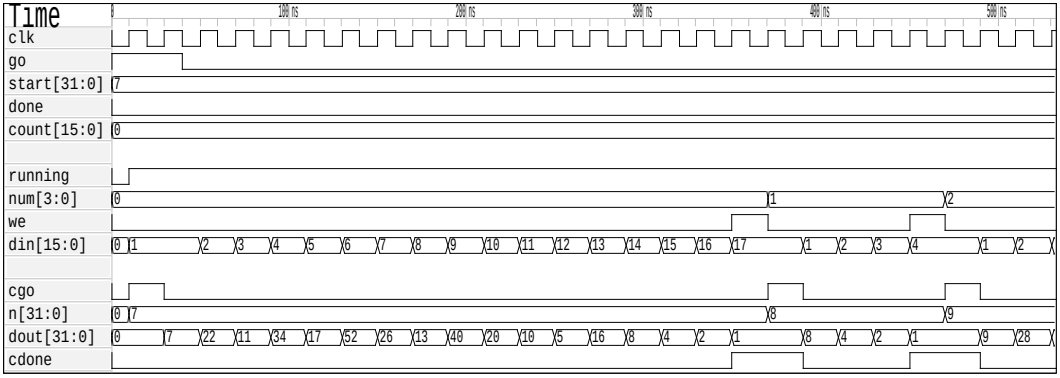
Once *done* is asserted, applying an address to *start* should read the memory from that address and present it on *count* in the next cycle.

Fill in the skeleton *range.sv* file provided. While you may modify anything you want except the interface to the module, we suggest you use the RAM and internal signals provided.

We have supplied a testbench file *range.cpp* that provides the clock, *go*, and *start* signals, then waits for *done* before reading out the number of iterations observed by applying different values of *start* to read the value out through the *count* signal.

As before, *make range.vcd* will compile the simulator, run the testbench, print the iteration counts that are written to memory, and write the *range.vcd* file.

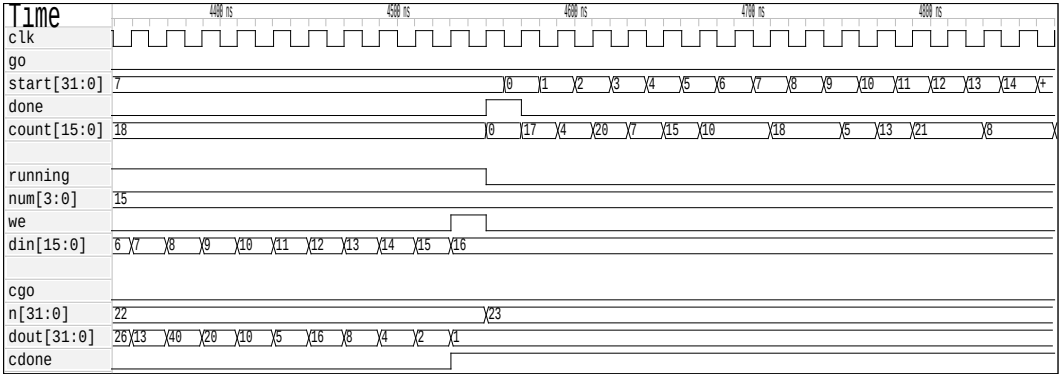
Below is the timing diagram of our solution as it starts; your solution only has to obey the protocol at the interface (clk, go, start, done, and count).



The *go* input switches *running* to true; loads *n* with the value on *start*; resets *num*, the RAM address, to 0; sets *din*, the iteration count, to 1; and pulses *cgo* high for a cycle to start the Collatz iterator module.

Every running cycle when *cgo* is low before *cdone* goes high, *din*, the number that will be ultimately written to the RAM, is increased by one. When the Collatz module asserts *cdone*, the *we* signal is pulsed high for just a single cycle to write the current count (*din*) to the RAM at the address in *num*. The cycle after *cdone* is asserted, *n* is increased by one, *din* is reset to 1, and *cgo* is asserted again to start testing the next value.

Below is the timing diagram of our solution as it finishes the Collatz iterations and switches to reading out the numbers it found.



After *we* is asserted for the highest RAM address (15, set by *RAM_WORDS*), *running* turns off and *done* pulses once. This tells the testbench to feed 0, 1, 2, … into *start* and the number of iterations for each value is read out on *count*. For example, *start* = 0 produces a 17 on *count*, corresponding to $n = 7$, and *start* = 11 produces a 21, corresponding to $n = 7 + 11 = 18$.

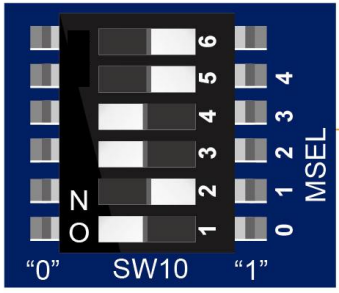
Running the *Vrange* simulator directly prints out these iteration counts:

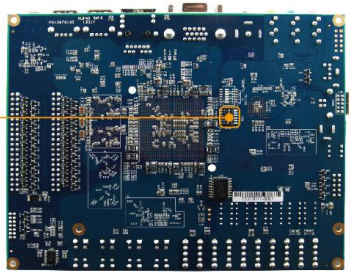
```
$ ./obj_dir/Vrange
7 17
8 4
9 20
10 7
11 15
12 10
13 10
14 18
15 18
16 5
17 13
18 21
19 21
20 8
21 8
22 16
```

Note that these counts match those listed earlier, although they were in hexadecimal.

6 Set the FPGA Configuration Mode

A microscopic set of switches on the back of the board controls the source of the FPGA's configuration information. For this lab, set it to the "Active Serial" mode as shown on the right.





For this lab, we will be configuring the FPGA with a JTAG interface through USB; the Active Serial mode makes the board start up in a factory demonstration mode.

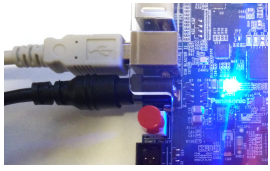
Mode	6	5	4	3	2	1
Active Serial (Default; use for this lab)	Off	Off	On	On	Off	On
FPPx16 (from SD card; later labs will use this)	Off	On	On	On	On	On
FPPx32 (from Linux)	Off	On	Off	On	Off	On

7 Compile and Download the Project Via the Command-Line

Enter the *lab1* directory and run *make lab1.qpf* to create the project from *de1-soc-project.tcl*. This should report "Info (23030): Evaluation of Tcl script de1-soc-project.tcl was successful," but if it complains "quartus_sh: Command not found," make sure your *PATH* variable includes the directory for the Quartus binaries (on the 1235 Mudd machines, */etc/profile.d/quartus.sh* does so when you log in). Running this script creates *lab1.qpf* (the main project file), *lab1.qsf* (settings, including files and pins), and *lab1.sdc* (clock constraints).

Once the project is created, you can compile it from the command-line with *make output_files/lab1.sof*. This reads the .sv files and ultimately produces the *lab1.sof* (SRAM object) file, which is downloaded to the FPGA to run your project. This takes a while, and will report and handful of warnings, but should eventually report "Info (293000): Quartus Prime Full Compilation was successful." You may ignore warnings "(292013) Feature LogicLock" and "(15714) Some pins have incomplete I/O assignments"; others should be fixed.

Connect the DE1-SoC board to your workstation. Connect the +12V power supply to the board near the red power button, connect a USB cable to the "USB Blaster" port on the board (next to the power button) and to your workstation, and power on the board with the red button.



Once your project is compiled, download the .sof file to the DE1-SoC board by running *make program*. A variety of things can go wrong. If you get "Error (213013): Programming hardware cable not detected," check your board's power and USB connection to the your workstation.

When powered and connected, the board should appear as a USB device. Under Linux, *lsusb* should report the board as 09fb:6810 Altera or 09fb:6010 Altera.

The Quartus software must also have permission to access the port. Run *jtagconfig*. With the board connected and powered on, it should report

```
$ jtagconfig
1) DE-SoC [1-1.5.2.2]
  4BA00477   SOCVHPS
  02D120DD   5CSE(BA5|MA5)/5CSTFD5D5/..
```

If *lsusb* "sees" the board but *jtagconfig* reports "No JTAG hardware available," there is a permission problem, which can be resolved by telling udev to make the board accessible to everybody. As root, create the file */etc/udev/rules.d/51-altera.rules* containing

```
ATTR{idVendor}=="09fb", ATTR{idProduct}=="6010", MODE="0666"
ATTR{idVendor}=="09fb", ATTR{idProduct}=="6810", MODE="0666"
```

8 Compile and Download the Project Via the GUI (optional)

The project can also be compiled and downloaded via the Quartus GUI. Start from a directory with a clean unpack of lab1.tar.gz (or run *make clean*), then start Quartus by typing *quartus*.

Create the project files by running a Tcl script. Open the Tcl console window with View→Utility Windows...→Tcl Console. Type *source de1-soc-project.tcl* in the Quartus Prime Tcl Console window. This will create the project files lab1.qpf, lab1.qsf, and lab1.sdc.

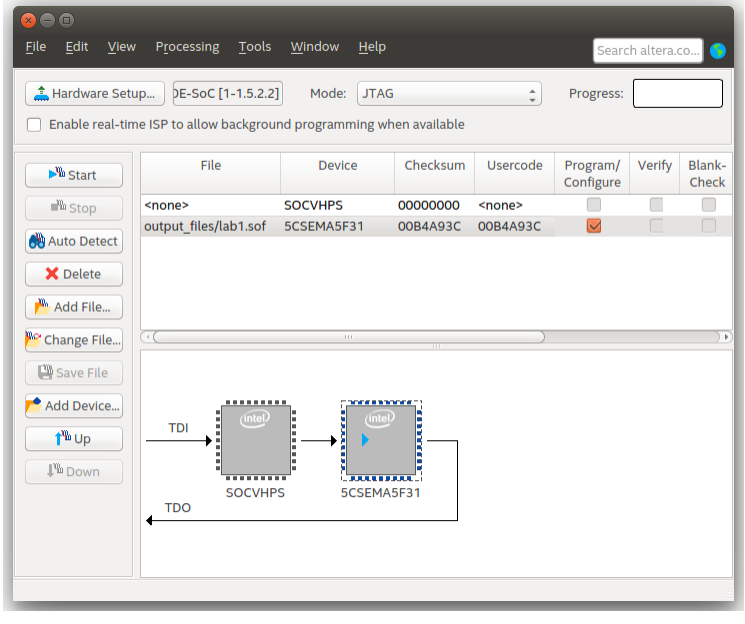
Open the *lab1.qpf* project with File→Open Project....

Compile the project with Processing→Start Compilation. This will take a while and should eventually report *Quartus Prime Full Compilation was successful.* There should be no errors, but there may be warnings.

Download the configuration to the FPGA. Select Tools→Programmer. If "No Hardware" appears, connect the board to your workstations via USB and power it on (see the previous section), then click on "Hardware Setup..." You should see "DE-SoC" under "Available hardware items." Select "De-SoC[···]" under "Currently selected hardware" and click "Close."

Set up the JTAG chain by clicking on "Auto Detect" and select "5CSEMA5." Answer "yes" if it asks to update the programmer's device list.

Tell it to configure the FPGA with the *lab1.sof* file by clicking on the "5CSEMA5" device then "Change File" and choose the *lab1.sof* file in the *output_files* directory. Mark the "Program/-Configure" checkbox on the 5CSEMA5F31 line. It should look like the image on the right.



Finally, click on "Start" to program the FPGA. This should quickly report "100% (Successful)" on the programmer.

9 Add a User Interface

Add a user interface that uses the four pushbuttons and the ten switches to test the number of iterations taken to reach 1 for various values of n. Modify the *lab1.sv* file we provided.

Have the ten switches SW[9:0] control the value n at the start of the range to test. Make the leftmost button KEY[3] run the *range* module over 256 values (i.e., trigger *go*) starting from the value on the switches (in binary).

Use your *hex7seg* module to make the leftmost three seven-segment displays show the value of n (specifically, the lower twelve bits) and have the rightmost three displays show the number of iterations taken to reach 1 for that value of n.

For example, if you enter 7 (in binary) on the switches and press KEY[3], the display should show 007011, which indicates $n = 7$ takes 17 iterations (in decimal).

Make it so that the rightmost buttons, KEY[0] and KEY[1] increment and decrement the value of n being displayed. Make it so holding them makes the value change about 5 times a second, e.g., by using a 22-bit counter running off the 50 MHz clock and only changing the value when this counter wraps around. The lowest n should always be set by the switches; the buttons should just control which number (between n and $n + 255$) is being read out.

Make it so KEY[2] (second to left) resets the difference between the n displayed and the value on the switches.

10 Demonstrate Your Working System

Every student needs to do a demonstration of his or her lab 1 design. The main objective of the demo is to test the user interface, so it will focus on the buttons, switches, and seven-segment displays. We will check the Collatz values from your submitted code.

You can demonstrate your working system during TA office hours.

We will check the following input and output during the demo:

- switch input
- seven-segment display output
- button input
  - regular button press
  - fast button press
  - slow button press
  - button press and hold
  - multiple button press
- indication that range is complete

This rubric deliberately does not specify exactly what your system should do in each of these cases because we want you to think about what "the right thing" is according to what a person would expect. For many of these actions there are multiple appropriate responses. Consider what you would expect from each action, and design your system accordingly. The TAs are happy to discuss what is "reasonable" behavior if you have questions.